Драйверы, соединения и запросы

API JDBC (*Java DataBase Connectivity*) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В настоящий момент JDBC выделены три типа драйверов:

1. Драйвер, представляющий собой частично библиотеку Java, работающий через *native* библиотеки для взаимодействия с клиентом СУБД.
2. Драйвер только на основе Java, работающий по сетевому и независимому от СУБД протоколу, который, в свою очередь, подключается к клиенту СУБД.
3. Сетевой драйвер, состоящий только из библиотеки Java, работающий напрямую с клиентом СУБД.

Если приложение выполняется на сервере, который не предполагает установки клиента СУБД, то выбор производится между вторым и третьим типами. Причем третий тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер третьего типа будет более эффективным с точки зрения производительности.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных, для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Последовательность действий для выполнения первого запроса.

1. *Подключение библиотеки с классом-драйвером базы данных.*

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, поместив ее предварительно в папку **/lib** приложения или сервера приложений.

mysql-connector-java-[*version***]-bin.jar** для СУБД MySQL,
ojdbc[*version*].**jar** для СУБД Oracle.

2. *Установка соединения с БД.*

До появления JDBC 4.0 объект драйвера СУБД для консольных приложений нужно было регистрировать с помощью вызова:

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver()); // for MySQL
DriverManager.registerDriver(new oracle.jdbc.OracleDriver()); // for Oracle
```

или создавать явно

```
Class.forName("com.mysql.cj.jdbc.Driver");
Class.forName("oracle.jdbc.OracleDriver");
```

В настоящее время в большинстве случаев в этом нет необходимости, так как экземпляр драйвера загружается автоматически при попытке получения соединения с БД.

Для установки соединения с БД вызывается один из перегруженных статических методов **getConnection()** класса **java.sql.DriverManager**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект **java.sql.Connection**. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов.

```
Connection connection = DriverManager.getConnection(
                              "jdbc:mysql://localhost:3306/testphones","root", "pass");
Connection connection = DriverManager.getConnection(
                        "jdbc:oracle:thin:@//localhost:1521:testphones", "system", "pass");
```

В результате будет возвращен объект **Connection** и создано одно установленное соединение с БД, именуемой **testphones**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода **getDrivers()**, **Stream<Driver> drivers()** можно получить список всех доступных драйверов.

3. *Создание объекта для передачи запросов.*

После создания объекта **Connection** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **java.sql.Statement**, создаваемый вызовом метода **createStatement()** класса **Connection**.

```
Statement statement = connection.createStatement();
```

Объект класса, реализующего интерфейс **Statement**, используется для прямого выполнения SQL-запроса. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур. Оба этих интерфейса наследуют возможности интерфейса **Statement**.

Метод **createStatement(int resultSetType, int resultSetConcurrency)** позволяет установить условия прокрутки и изменения объекта **ResultSet**.

Параметр **resultSetType** со значением **ResultSet**.**TYPE_FORWARD_ONLY** позволяет продвигаться по объекту только от начала к концу и выставляется по умолчанию. Значение **ResultSet**.**TYPE_SCROLL_INTENSIVE** разрешает навигацию в обе стороны и не учитывает изменения от других пользователей.

ResultSet.**TYPE_SCROLL_SENSITIVE** разрешает навигацию в обе стороны и учитывает изменения от других пользователей после того, как **ResultSet** был получен.

Значение **ResultSet.CONCUR_READ_ONLY** параметра **resultSetConcurrency** позволяет только читать результат и устанавливается по умолчанию. Значение **ResultSet.CONCUR_UPDATABLE** создает **ResultSet** с возможностью изменения данных.

4. *Выполнение запроса.*

Созданный объект **Statement** можно использовать для выполнения запросов SQL, передавая их в один из методов:

ResultSet executeQuery(String sql) — выполняет запросы **SELECT**. Результаты выборки из базы помещаются в объект **ResultSet**:

int executeUpdate(String sql) — выполняет запросы, изменяющие состояние базы **INSERT**, **UPDATE**, **DELETE**. Возвращает количество строк, задействованных запросом;

boolean execute(String sql) — применяется для выполнения произвольных запросов;

int[] executeBatch() — выполняет *batch*-команды, т.е группу запросов, как один запрос

```
// extract all data from the phonebook table
ResultSet resultSet = statement.executeQuery(
                                        "SELECT idphonebook, lastname, phone FROM phonebook");
```

5. Обработка результатов запроса на выборку данных производится методами интерфейса **ResultSet**, где самыми распространенными являются **next()**, **first()**, **previous()**, **last()**, **beforeFirst()**, **afterLast()**, **isFirst()**, **isLast()**, **absolute(int i)** — методы навигации по строкам таблицы результатов, группа методов по доступу к информации по номеру позиции в записи вида **String getString(int pos)**, а также аналогичные методы, начинающиеся с **get***Type***(int pos)** (**int getInt(int pos)**, **float getFloat(int pos)** и др.) и **update***Type***()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, позволяющие извлекать из полей таблицы специфические объекты (*Character Large Object*, *Binary Large Object*), которые могут быть, например, графическими или архивными файлами.

Следует обратить внимание, что счет позиций в **ResultSet** начинается с **«1»**, а не с **«0»**, как в коллекциях и массивах.

Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа **int getInt(String columnLabel)**, **String getString(String columnLabel)**, **Object getObject(String columnLabel)** и подобными им.

Обновляемый набор данных позволяет обновлять, изменять и **ResultSet**, и информацию в таблице базы данных: **updateRow()**, **insertRow()**, **updateString()** и др.

При первом вызове метода **next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

6. *Закрытие соединения.*

```
connection.close(); // closes also Statement & ResultSet
```

Когда база больше не нужна, соединение должно быть закрыто. Для того, чтобы правильно пользоваться приведенными методами, программисту как минимум требуется знать SQL, способ организации конкретной БД, типы полей БД и др.

7. *Выгрузка драйверов.*

По завершении работы приложения следует выгрузить или дерегистрировать драйвер:

```
DriverManager.getDrivers().asIterator().forEachRemaining(driver -> {
  try {
    DriverManager.deregisterDriver(driver);
  } catch (SQLException e) {
    // log
  }
});
```

Соединение и запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **testphones** и одной таблицей **PHONEBOOK**. Таблица должна содержать три поля: числовое (первичный ключ) — **IDPHONEBOOK**, символьное — **LASTNAME** и числовое — **PHONE** и несколько занесенных записей.

IDPHONEBOOK	LASTNAME	PHONE
1	Руденко	7756544
2	Artukevich	6861880

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение любых символов.

Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы:

/ # 1 # соединение с БД и простой запрос # SimpleJdbcMain.java */*

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
public class SimpleJdbcMain {
  public static void main(String[] args) {
    try {
      DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
    } catch (SQLException e) {
      e.printStackTrace();
    }
    String url = "jdbc:mysql://localhost:3306/testphones";
    Properties prop = new Properties();
    prop.put("user", "root");
    prop.put("password", "pass»);
    prop.put("autoReconnect", "true");
    prop.put("characterEncoding", "UTF-8");
    prop.put("useUnicode", "true");
    prop.put("useSSL", "true");
    prop.put("useJDBCCompliantTimezoneShift", "true");
    prop.put("useLegacyDatetimeCode", "false");
    prop.put("serverTimezone", "UTC");
    prop.put("serverSslCert", "classpath:server.crt");
    try (Connection connection = DriverManager.getConnection(url, prop);
         Statement statement = connection.createStatement()) {
      String sql = "SELECT idphonebook, lastname, phone FROM phonebook";
      ResultSet resultSet = statement.executeQuery(sql);
      List<Abonent> abonents = new ArrayList<>();
      while (resultSet.next()) {
        int id = resultSet.getInt(1);
        String name = resultSet.getString(2);
        int number = resultSet.getInt("phone");
        abonents.add(new Abonent(id, name, number));
      }
      System.out.println(abonents);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

В простом приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие СУБД. Объект **ResultSet** также нуждается в закрытии, но его автоматически закрывает метод **close()** интерфейса **Statement** при закрытии или механизм **AutoCloseable**.

Класс **Abonent**, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

/ # 2 # классы с информацией # Entity.java # Abonent.java */*

```
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {
}
```

```
public class Abonent extends Entity {
  private int id;
  private String name;
  private int phone;
  public Abonent() {
  }
  public Abonent(int id, String name, int phone) {
    this.id = id;
    this.name = name;
    this.phone = phone;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getPhone() {
    return phone;
  }
  public void setPhone(int phone) {
    this.phone = phone;
  }
  public String toString() {
    final StringBuilder sb = new StringBuilder("Abonent{");
    sb.append("id=").append(id).append(", name='").append(name).append('\'');
    sb.append(", phone=").append(phone).append('}');
    return sb.toString();
  }
}
```

Параметры соединения можно задавать: с помощью прямой передачи значений в коде класса, а также с помощью файлов **properties**, **json**, **yaml** или **xml**. Окончательный выбор производится в зависимости от конфигурации проекта.

Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Пусть класс **ConnectorCreator** использует файл ресурсов **database.properties**, в котором хранятся, как правило, такие параметры подключения к БД, как логин, пароль доступа и др.

```
db.driver=com.mysql.cj.jdbc.Driver
user     = root
password = pass
poolsize = 32
db.url  = jdbc:mysql://localhost:3306/testphones
useUnicode = true
encoding = UTF-8
useSSL   = true
useJDBCCompliantTimezoneShift = true
useLegacyDatetimeCode = false
serverTimezone = UTC
serverSslCert = classpath:server.crt
```

Код класса **ConnectionCreator** может выглядеть следующим образом:

```
/* # 3 # создание соединений с БД # ConnectionCreator.java */
```

```
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
public class ConnectionCreator {
  private static final Properties properties = new Properties();
  private static final String DATABASE_URL;
  static {
    try {
      properties.load(new FileReader("datares\\database.properties"));
      String driverName = (String) properties.get("db.driver");
      Class.forName(driverName);
    } catch (ClassNotFoundException | IOException e) {
      e.printStackTrace(); // fatal exception
    }
    DATABASE_URL = (String) properties.get("db.url");
  }
  private ConnectionCreator() {}
  public static Connection createConnection() throws SQLException {
    return DriverManager.getConnection(DATABASE_URL, properties);
  }
}
```

В таком случае получение соединения с БД сведется к вызову:

```
Connection connection = ConnectionCreator.createConnection();
```

Класс **ConnectorCreator** лучше сделать синглтоном.

Добавление и изменение записи

Объект **ResultSet** позволяет вставлять запись в базу данных без дополнительных запросов. Объект **Statement** нужно создавать с разрешением на изменение в базе данных.

```
  try (Connection connection = DriverManager.getConnection(url, prop);
       Statement statement = connection.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
     ResultSet resultSet = statement.executeQuery(
             "SELECT idphonebook, lastname, phone FROM phonebook");
     resultSet.moveToInsertRow();  // insert row
     resultSet.updateInt(1, 77);
     resultSet.updateString(2, "Bahdanovich");
     resultSet.updateInt(3, 222322);
     resultSet.insertRow();
     resultSet.moveToCurrentRow();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Изменения в базу данных также легко вносятся с помощью возможностей **ResultSet**:

```
while (resultSet.next()) {
  int id = resultSet.getInt(1);
  if (id == 2) {
    resultSet.updateInt("phone", 550055); // update row
    resultSet.updateRow();
  }
}
```

В результате у записи с **IDPHONEBOOK=2** номер телефона будет заменен как в **ResultSet**, так и в базе данных.

Метаданные

Существует целый ряд методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData** для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для записей подобных методов нет.

Получить объект **ResultSetMetaData** можно следующим образом:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

Некоторые методы интерфейса **ResultSetMetaData**:

int getColumnCount() — возвращает число столбцов набора результатов объекта **ResultSet**;

String getColumnName(int column) — возвращает имя указанного столбца;

String getColumnTypeName(int column) — возвращает тип данных указанного столбца.

Если добавить следующий код к примеру #1

```
System.out.println("ColumnCount: " + rsMetaData.getColumnCount());
System.out.println("ColumnName: " + rsMetaData.getColumnName(1));
System.out.println("ColumnTypeName: " + rsMetaData.getColumnTypeName(1));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(1));
System.out.println("ColumnName: " + rsMetaData.getColumnName(2));
System.out.println("ColumnTypeName: " + rsMetaData.getColumnTypeName(2));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(2));
```

то в консоль будет выведено:

ColumnCount: 3
ColumnName: idphonebook
ColumnTypeName: INT
isAutoIncrement: true
ColumnName: lastname
ColumnTypeName: VARCHAR
isAutoIncrement: false

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

Некоторые методы обширного интерфейса **DatabaseMetaData**:

String getDatabaseProductName() — возвращает название СУБД;

String getDatabaseProductVersion() — номер версии СУБД;

String getDriverName() — имя драйвера JDBC;

String getUserName() — имя пользователя БД;

String getURL() — местонахождение источника данных;

ResultSet getTables() — набор типов таблиц, доступных для данной БД.

Если добавить следующий код к примеру #1

```
System.out.println("DatabaseName: " + dbMetaData.getDatabaseProductName());
System.out.println("DatabaseVersion: " + dbMetaData.getDatabaseProductVersion());
System.out.println("UserName: " + dbMetaData.getUserName());
System.out.println("URL: " + dbMetaData.getURL());
```

то в консоль будет выведено:

DatabaseName: MySQL
DatabaseVersion: 5.7.15-log
UserName: root@localhost
URL: jdbc:mysql://localhost:3306/testphones